# AndroidCompass: A Dataset of Android Compatibility Checks in Code Repositories

Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, Frank Ortmeier
Otto-von-Guericke University Magdeburg, Germany
{sebastian.nielebock | paul.blockhaus | jacob.krueger | frank.ortmeier}@ovgu.de

***Abstract*—Many developers and organizations implement apps for Android, the most widely used operating system for mobile devices. Common problems developers face are the various hardware devices, customized Android variants, and frequent updates, forcing them to implement workarounds for the different versions and variants of Android APIs used in practice. In this paper, we contribute the *Android Compatibility checkS dataSet* (AndroidCompass) that comprises changes to compatibility checks developers use to enforce workarounds for specific Android versions in their apps. We extracted 80,324 changes to compatibility checks from 1,394 apps by analyzing the version histories of 2,399 projects from the F-Droid catalog. With AndroidCompass, we aim to provide data on when and how developers introduced or evolved workarounds to handle Android incompatibilities. We hope that AndroidCompass fosters research to deal with version incompatibilities, address potential design flaws, identify security concerns, and help derive solutions for other developers, among others—helping researchers to develop and evaluate novel techniques, and Android app as well as operating-system developers in engineering their software.**

***Index Terms*—Android, compatibility, API, dataset**

## I. Introduction

Android is the prevailing operating system for mobile devices worldwide,[1] with many developers working on Android itself, vendor-specific customizations, and mobile apps for users. Since September 2008, Android has been released in 11 major releases with 30 different Android API levels (as of January 2021). Consequently, most developers have to react regularly to version updates of Android, which, for instance, add support for new sensors and devices, address security concerns, or replace functionality. While such updates promise improvements, not all vendors and users update their Android system, for example, because they first need to adapt their own customizations or because the hardware does not support the Android version anymore. As a result, Android exists in different versions, each with API-specific functionalities that are not available in other versions.

Thus, Android apps face compatibility issues, for instance, because they rely on functionality that is deprecated in a new release. Developers have to handle such incompatibilities to make their apps available for as many devices as possible—which, however, is challenging as recently shown again for COVID-19 tracing apps [1]. To deal with incompatibilities, Android enables developers to configure their apps for specific API levels and provide alternative implementations. The latter is typically achieved by checking the Android version of the device and adapting the control flow accordingly.

Recent research [5], [10], [21]–[23] has studied the prevalence of and fixing strategies for such incompatibilities. However, except for the work of Scalabrino et al. [15], [16], no research has been conducted on how Android incompatibilities are handled during the evolution of an app and Android itself. Such evolution is particularly interesting, since it allows researchers to investigate *when* and *in what order* incompatibilities were handled with what workarounds.

In this paper, we describe AndroidCompass, a dataset of historical changes of compatibility checks in the source code of Android apps. AndroidCompass comprises 80,324 individual single-line code changes of Android compatibility checks and their respective meta data, which we collected from 1,394 projects of the F-Droid catalog.[2] We hope to foster research on how developers introduce, change, and eventually fix compatibility checks, ideally leading to automatic support for suggesting fixing patterns to developers. Our dataset and all artifacts related to this paper are publicly available.[3]

## II. Android Compatibility Issues

The variety of functionalities available for different devices as well as variants of the Android API causes compatibility issues in apps. In a general sense, an *incompatibility* (or compatibility issue) refers to a "state of not being able to exist or work with another person or thing because of basic differences."[4] In our case, an Android incompatibility refers to the use of an Android API element (e.g., a method) in an app while that element is not available (or behaviorally equal) in the Android API level of the underlying device. Note that Android versions, such as KITKAT or OREO, may consist of multiple Android API levels, usually denoted by increasing integers for newer levels.

Scalabrino et al. [15], [16] distinguish between *forward* and *backward* incompatibility from the perspective of an app. Forward incompatibility occurs when the app is used with a newer version (i.e., higher API level) and an API element becomes incompatible (e.g., the element is removed in the newer API level). Backward incompatibility occurs when the app is used with an older version (i.e., lower API level) and an API element becomes incompatible (e.g., it was not yet implemented in the older API level). The Android

[1]https://gs.statcounter.com/os-market-share/mobile/worldwide

[2]https://f-droid.org/en/

[3]https://doi.org/10.5281/zenodo.4428340

[4]https://dictionary.cambridge.org/us/dictionary/english/incompatibility

```
if(Build.VERSION.SDK_INT < Build.VERSION_CODES.KITKAT){
 //call API before Kitkat
} else {
 //call API of Kitkat
}
```

Listing 1. An in-code Android compatibility check.

framework usually avoids (with some rare exceptions) forward incompatibility by not deleting, but deprecating, obsolete API elements—which is denoted backward compatibility from the perspective of the Android framework. This claim has been supported by recent research. For example, He et al. [5] have been able to execute 4,041 out of 4,697 apps on newer Android versions without any modifications.

App developers can handle incompatibilities by configuring the minimally and maximally allowed Android API levels in the `AndroidManifest.xml`. However, configuring the maximal level is neither enforced nor recommended, since the Android framework usually avoids forward incompatibilities.[5] By defining the range of allowed API levels, an app cannot be installed on devices using an API level outside of this range. Still, since this range may be too coarse-grained, app developers frequently check the actual Android version in the code itself [5], [15], [16]. For example, the code in Listing 1 checks whether the Android API level (i.e., `Build.VERSION.SDK_INT`) is below the predefined constant for KITKAT (i.e., `Build.VERSION_CODES.KITKAT`) and changes the control flow accordingly. Note that Xia et al. [23] have found that only ≈38 % of such checks provide an actual alternative functionality, while most checks only disable functionality. To avoid backward incompatibilities, Android provides newer functionalities with a support library since API level 26,[6] and since API level 28 this library is part of the Android JETPACK libraries denoted as ANDROIDX.[7] Unfortunately, this support library only provides a subset of newer API functionalities.

Recent research focused on the prevalence of Android incompatibilities and tool support to detect these automatically, for instance, with FicFinder [21], [22], CiD [10], IctAPIFinder [5], ACRyL [15], [16], and RAPID [23]. Essentially, all these tools identify the used Android API elements (typically method calls) and map them to the API levels in which they are available. Afterwards, they check whether these API levels are within those allowed for the app (i.e., in the `AndroidManifest.xml`) and whether the usage is protected with conditional statements (e.g., `if`). Since these tools require control-flow information, they analyze the byte code of the app (i.e., the `*.apk`). The results indicate that Android incompatibilities are prevalent with ≈25 % to ≈83 % of the analyzed apps having at least one compatibility issue [5], [15], [16], [23]. Root causes for these incompatibilities are device-specific reasons (e.g., different hardware, customized operating system) as well as Android-specific reasons (e.g., API evolution with insufficient support, errors in the Android API) [5], [16], [21], [22]. Mostly, the identified incompatibilities caused functional problems with the app deviating from its intended behavior, but also problems with the performance, security, or user experience [16], [21], [22].

The datasets used for those analyses mostly refer only to the latest version of the app at the time of the analysis. While this is sufficient to elaborate on the prevalence of incompatibilities, it does not help to shed light on the learning curve and evolution of compatibility handling in Android. For example, it remains unclear how long it takes developers to identify and fix incompatibilities with new Android versions and how frequently version checks are changed until they are in a robust state or replaced. The only exception is the work of Scalabrino et al. [15], [16] who considered 19,291 snapshots of 1,170 apps. While having a comparable size to AndroidCompass, using the data of Scalabrino et al. requires parsing and compiling of source code. This is not necessary for AndroidCompass, since it requires textual filtering only. Moreover, AndroidCompass has more fine-grained timestamps based on commit information, while Scalabrino et al. used pre-sampled timestamps that are far more fragmented. Finally, AndroidCompass is easier to access and transfer, since it is a simple csv file.

## III. ANDROIDCOMPASS

Nest, we describe our dataset and its construction process.

### A. Dataset Construction

Initially, we conducted a preliminary analysis of Android-specific API misuses, which, except for the web crawling step, was independent of the construction of AndroidCompass. However, we derived our validation data from that analysis. In this context, we denote an API misuse as a deviant use of an API from the one that was intended by the developers, and that eventually leads to negative behavior of the software (e.g., a software crash or a performance issue) [2], [13], [14].

At first, we manually analyzed commits that fixed API misuses in FOSS Android apps, which we collected from F-Droid.[2] For this purpose, we implemented a web crawler using scrapy[8] to identify all URLs directing to a Git repository on GitHub, resulting in 2,399 initial repositories. We restricted our analysis to Git and GitHub, since these are the prevalent version-control and repository-hosting systems, respectively [7].

To identify commits that involved a fix, we used PyDriller [18] and extracted from the repositories all commits with a message containing the keywords "fix," "issue," or "bug." These keywords were inspired by previous work [17]. We reduced the number of commits to make them manually assessable. Particularly, we ignored repositories that excessively used these keywords in their messages (i.e., $>$6 % of all messages for repositories with $>$1,000 commits, or $>$10 % otherwise). Then, we extracted for each commit all lines with third-party API changes using our extraction mechanism [14]. Also, we considered only small commits that changed at most ten methods and that changed imports prefixed with `android.*`. Finally, we transformed commit messages with standard natural-language processing techniques and filtered

[5]https://developer.android.com/guide/topics/manifest/uses-sdk-element
[6]https://developer.android.com/topic/libraries/support-library
[7]https://developer.android.com/jetpack/androidx

[8]https://scrapy.org/

```
^(([^\*/]*VERSION.SDK_INT[ ]+(<|<=|==|>=|>).*) |
([^\*/]*(<|<=|==|>=|>)[ ]+[a-zA-Z0-9\.]*VERSION.SDK_INT.*))
```

Listing 2. Our regular expression to identify Android compatibility checks.

whether they contain "android," "api level," or any of the Android version names.[9] We derived these steps in an exploratory manner to construct a small, relevant, and manually assessable validation dataset (of limited generalizability).

Then, the first three authors assessed the remaining 522 commits from 278 repositories independently to identify fixing commits of Android API misuses. We determined that 132 of them represented API-misuse fixes. Aligning to previous studies, we found that compatibility issues were prevalent among these misuses. As a result, we marked each commit considering whether they added or changed a compatibility check (cf. Listing 1). We found that 67 of the 132 commit changes involved code for handling compatibility issues.

Based on our manual assessment, we derived the regular expression we show in Listing 2 to automatically detect compatibility checks against the `VERSION.SDK_INT`. This expression is part of a python script within our replication package.[3] Again, we used PyDriller to identify changed lines together with a set of meta-information (cf. Table I) for each line of code matching the regular expression.

We intended AndroidCompass to provide information regarding which Android version was checked. For this purpose, we extracted the version to which the `VERSION.SDK_INT` constant is compared to and a normalized comparison type (i.e., `<|<=|==|>=|>` with the actual API level on the left). To refine our dataset, we normalized the version codes, since developers can represent those, for instance, as numbers (e.g., 19), constants (e.g., `Build.VERSION_CODES.KITKAT`), or results of a method call (e.g., `getKitkatCode()`). We transformed numbers to version constants based on the Android documentation[10] (e.g., 19 refers to `Build.VERSION_CODES.KITKAT`). Similarly, we normalized varying version constants (e.g., `android.os.Build.VERSION_CODES.KITKAT`) into that same format. We marked versions as `NONE_DETECTED` if we could not reliably ensure the correct version (e.g., in case of user-defined constants/methods or multiple version checks in a single line). However, such corner cases occur sparsely within AndroidCompass (≈2.3 %).

We validated our analysis script based on the 132 commits we previously identified to comprise fixes for API misuses. At this point, we denoted a detection as false negative if we assessed a commit to contain a compatibility check, but our script did not. Vice versa, we denoted a detection as false positive if we did not identify a compatibility check, but our script did. At first, we achieved a precision of 98 % and a recall of ≈73.1 %. While validating these results, we found that the single false positive was actually a mistake in our manual assessment, meaning that the precision was 100 %. Regarding false negatives, in most cases, PyDriller did not find the respective commit. In other cases, our regular expression did not match, for instance, if the `VERSION.SDK_INT` constant was stored in a local variable or if exceptions of the API were handled in an inherited, customized class. We found two more cases that revealed errors in our regular expressions, which we fixed accordingly. After re-validating, we achieved an improved recall of ≈76.1 % with the same precision of 100 %.

Finally, we analyzed whether the extracted data is reasonable by comparing the proportion of lines adding a compatibility check to all compatibility checks that were changed in a project:

$$\frac{\#added}{\#added + \#deleted}$$

We expected each project to have a value ≥0.5, since one cannot delete more lines than have been added. However, for one project this value was below 0.5. We found that this was caused by Git's branching mechanism: The developers created two separate branches, removed the same compatibility checks in both, and merged them. As a result, the merged history contains multiple commits deleting the same compatibility checks. Thus, we cannot ensure that the actual number of compatibility checks is the balance of added and deleted checks. Still, for all other projects, our assumption held true, supporting our confidence in the data. Moreover, AndroidCompass does not contain the respective guarded method calls, and thus additional analyses are required (e.g., control-flow analysis). In our replication package, we provide a script using a regular expression to extract the code section succeeding the compatibility check. Since it is a coarse method and to avoid copyright and licensing issues [3], [4], the extracted sections are not part of the dataset.

After this validation, we executed our script on all 2,399 Git repositories we crawled. We inferred commits until the `author_date` timestamp of January 6th, 2021 (11:59:59 pm AoE). Our script required roughly one day and we obtained 80,324 individually changed compatibility checks from 21,658 commits of 1,394 repositories. We could extract Android versions and respective comparison operators for 78,503 (≈97.7 %) of these compatibility checks.

## B. Dataset Description

AndroidCompass is a single csv file containing all changed compatibility checks as well as metadata of the commit and code. We summarize the individual fields in Table I. For replication, we publish all of our artifacts.[3]

We denote whether the line containing the compatibility check was added (+++) or deleted (−−−) in the field `change_action`. If the compatibility check was changed (i.e., the code was modified), at least two entries (an addition and a deletion) for the same commit hash and source file (`new_path`) exist. Comparing `old_path` and `new_path` allows to determine whether an addition or deletion is a result of a new, removed, or renamed file. The field `compare_type` represents the normalized operator (i.e., `{<|<=|==|>=|>}`), whereas the field `version` is the extracted version code. We further provide the `timestamp` of each commit based

[9]https://en.wikipedia.org/wiki/Android_version_history

[10]https://developer.android.com/reference/android/os/Build.VERSION_CODES

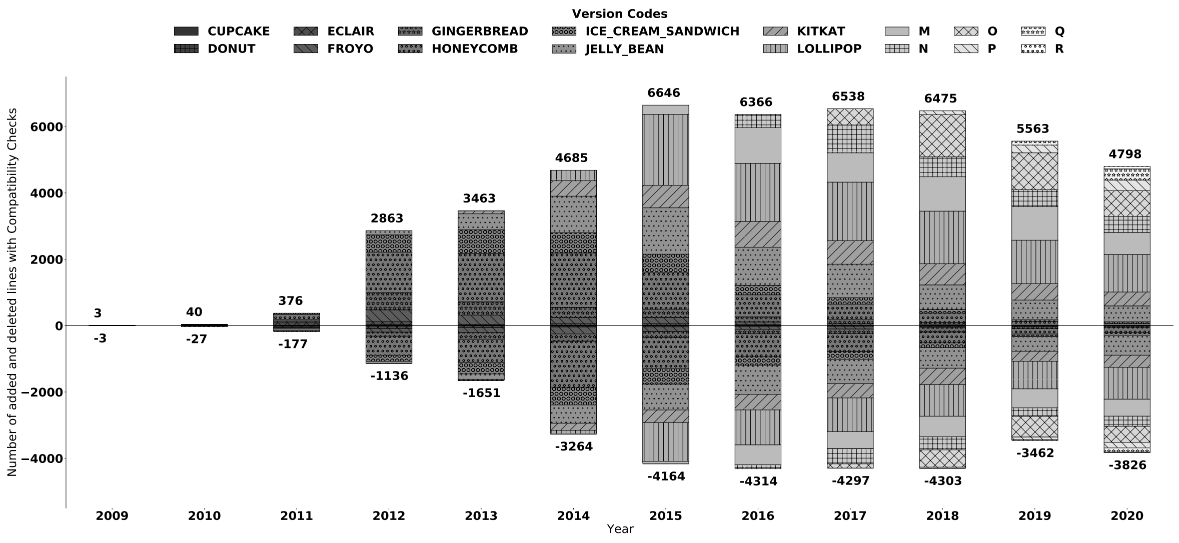

Fig. 1. A preliminary analysis of the historical development of added/deleted compatibility checks from 2009 to 2020.

TABLE I
FIELDS OF ANDROIDCOMPASS.

| field | description |
|---|---|
| `repo_name` | name of the repository |
| `repo_url` | URI of the repository |
| `commit_hash` | hash of the commit |
| `timestamp` | timestamp of the commit (i.e., the author date and time) |
| `old_path` | the old path of the changed file (empty if the file was added in the respective commit) |
| `new_path` | the new path of the changed file (empty if the file was deleted in the respective commit) |
| `change_action` | indicator whether the compatibility check was added (i.e., +++) or deleted (i.e., ---) |
| `line` | the line containing the compatibility check |
| `version` | the extracted version code from the `andorid.os.Build.VERSION_CODES`-package or `NONE_DETECTED` |
| `compare_type` | comparison used for the compatibility check (i.e. < \| <= \| == \| >= \| > or `NONE_DETECTED`) |
| `timestamp_ign_tz` | same as `timestamp`, but drop timezone |

on the local time with and without timezone information (i.e., `timestamp_ign_tz`). To not violate any developer's privacy, we deliberately left out any further information related to them (e.g., e-mail address, name, commit message). As long as the repositories remain publicly available, researchers can analyze such and additional information using the repositories' Git commit hashes (`commit_hash`).

## IV. CONCLUSION

With AndroidCompass, we aim to foster research on analyzing, detecting, and correcting Android incompatibilities. Since AndroidCompass comprises historical and evolutionary information, it allows researchers to investigate change patterns of compatibility checks. As an example of such an analysis, we display a first overview of the evolution of the compatibility checks in Figure 1. To simplify this overview, we merged the API levels into their respective versions, for instance, ECLAIR_0_1 and ECLAIR_MR1 become ECLAIR. We can see that certain versions, namely, HONEYCOMB, LOLLIPOP, MARSHMALLOW (M), and OREO (O), are involved in more changed compatibility checks than other versions. For example, even in 2020, compatibility checks for HONEYCOMB are changed, even though it is not maintained anymore since 2016.

Building on AndroidCompass, we envision several directions for future research, such as:

**Pattern Detection.** Detecting incompatibilities is a helpful means to improve the quality of software and avoid design flaws. Deriving patterns (i.e., similar to error patterns of Livshits and Zimmermann [11]) can help to improve such detection in an automated fashion, not only in the current code base, but also throughout a system's history. For instance, it has been shown that compatibility checks can help to detect device-induced [20] or callback-caused issues [6]. Such research helps to understand incompatibilities, their evolution, and impact, as well as the design of novel techniques.

**Automated Program Repair.** In general, Android incompatibilities can be considered as a form of API misuse. We may be able to use and cluster identified patterns (i.e., similar compatibility checks) to automatically identify design flaws in the code (e.g., security issues due to unguarded API calls) [14], [24]. Moreover, we can use the historical information to identify past fixes of incompatibilities and reuse them to automatically repair the same incompatibilities in other code locations, similar to research on automated bug repair [8], [9], [12], [19].

**Benchmarking.** Finally, AndroidCompass can serve as a dataset for benchmarking new tools, for instance, for automated program repair. Novel tools may exploit different techniques that can be evaluated and compared based on AndroidCompass. As a concrete example, we are working on a technique for cooperative program repair [13], for which we aim to use AndroidCompass to evaluate its performance.

**Acknowledgments.** This research has been supported by the German Research Foundation (SA 465/49-3).

## References

[1] N. Ahmed, R. A. Michelin, W. Xue, S. Ruj, R. Malaney, S. S. Kanhere, A. Seneviratne, W. Hu, H. Janicke, and S. K. Jha, "A survey of COVID-19 contact tracing apps," *IEEE Access*, vol. 8, pp. 134 577–134 601, 2020, https://doi.org/10.1109/ACCESS.2020.3010226.

[2] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static API-misuse detectors," *IEEE Transactions on Software Engineering (TSE)*, vol. 45, no. 12, pp. 1170–1188, 2019, https://doi.org/10.1109/TSE.2018.2827384.

[3] M. Ballhausen, "Free and open source software licenses explained," *IEEE Computer*, vol. 52, no. 6, pp. 82–86, 2019, https://doi.org/10.1109/MC.2019.2907766.

[4] S. Baltes and S. Diehl, "Usage and attribution of Stack Overflow code snippets in GitHub projects," *Springer Empirical Software Engineering (EMSE)*, vol. 24, no. 3, pp. 1259–1295, 2019, https://doi.org/10.1007/s10664-018-9650-5.

[5] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in Android apps," in *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*. ACM/IEEE, 2018, pp. 167–177, https://doi.org/10.1145/3238147.3238185.

[6] H. Huang, L. Wei, Y. Liu, and S.-C. Cheung, "Understanding and detecting callback compatibility issues for Android applications," in *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*. ACM/IEEE, 2018, pp. 532–542, https://doi.org/10.1145/3238147.3238181.

[7] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the 11th International Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 92–101, https://doi.org/10.1145/2597073.2597074.

[8] S. Kim, K. Pan, and E. E. J. Whitehead, "Memories of bug fixes," in *Proceedings of the 14th International Symposium Foundations of Software Engineering (FSE)*. ACM, 2006, pp. 35–45, https://doi.org/10.1145/1181775.1181781.

[9] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224, https://doi.org/10.1109/SANER.2016.76.

[10] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "CiD: Automating the detection of API-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018, pp. 153–163, https://doi.org/10.1145/3213846.3213857.

[11] B. Livshits and T. Zimmermann, "DynaMine: Finding common error patterns by mining software revision histories," in *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference/-Foundations of Software Engineering (ESEC/FSE)*. ACM, 2005, pp. 296–305, https://doi.org/10.1145/1081706.1081754.

[12] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 298–312, https://doi.org/10.1145/2837614.2837617.

[13] S. Nielebock, R. Heumüller, J. Krüger, and F. Ortmeier, "Cooperative API misuse detection using correction rules," in *Proceedings of the 42nd International Conference on Software Engineering - New Ideas and Emerging Results (ICSE-NIER)*. ACM/IEEE, 2020, pp. 73–76, https://doi.org/10.1145/3377816.3381735.

[14] S. Nielebock, R. Heumüller, K. M. Schott, and F. Ortmeier, "Guided pattern mining for API misuse detection by change-based code analysis," 2020, preprint. [Online]. Available: https://arxiv.org/abs/2008.00277

[15] S. Scalabrino, G. Bavota, M. Linares-Vásquez, M. Lanza, and R. Oliveto, "Data-driven solutions to detect API compatibility issues in Android: an empirical study," in *Proceedings of the 16th International Workshop on Mining Software Repositories (MSR)*. IEEE/ACM, 2019, pp. 288–298, https://doi.org/10.1109/MSR.2019.00055.

[16] S. Scalabrino, G. Bavota, M. Linares-Vásquez, V. Piantadosi, M. Lanza, and R. Oliveto, "API compatibility issues in Android: Causes and effectiveness of data-driven detection techniques," *Springer Empirical Software Engineering (EMSE)*, vol. 25, no. 6, pp. 5006–5046, 2020, https://doi.org/10.1007/s10664-020-09877-w.

[17] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR)*. ACM, 2005, pp. 1–5, https://doi.org/10.1145/1083142.1083147.

[18] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 26th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 908–911, https://doi.org/10.1145/3236024.3264598.

[19] B. Sun, G. Shu, A. Podgurski, S. Li, S. Zhang, and J. Yang, "Propagating bug fixes with fast subgraph matching," in *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2010, pp. 21–30, https://doi.org/10.1109/ISSRE.2010.36.

[20] L. Wei, Y. Liu, and S. Cheung, "PIVOT: Learning API-device correlations to facilitate android compatibility issue detection," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2019, pp. 878–888, https://doi.org/10.1109/ICSE.2019.00094.

[21] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2016, pp. 226–237, https://doi.org/10.1145/2970276.2970312.

[22] L. Wei, Y. Liu, S.-C. Cheung, H. Huang, X. Lu, and L. Xuanzhe, "Understanding and detecting fragmentation-induced compatibility issues for Android apps," *IEEE Transactions on Software Engineering (TSE)*, vol. 46, no. 11, pp. 1176–1199, 2020, https://doi.org/10.1109/TSE.2018.2876439.

[23] H. Xia, Y. Zhang, Y. Zhou, X. Chen, Y. Wang, X. Zhang, S. Cui, G. Hong, X. Zhang, M. Yang *et al.*, "How Android developers handle evolution-induced API compatibility issues: a large-scale study," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2020, pp. 886–898, https://doi.org/10.1145/3377811.3380357.

[24] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2009, pp. 318–343, https://doi.org/10.1007/978-3-642-03013-0_15.